SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 281–294 (1996)

Research

A Risk Index for Software Producers

HARETON K. N. LEUNG
Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

SUMMARY

Software risk management has been receiving increasing attention lately. A key activity of software risk management is the quantification of the risk of using a software product. Previous attempts to define the risk index have taken the users' viewpoint. The loss due to a product failure is computed based on its impact on the users. Yet, the impact of a failure on the software producer is different from that of the users. A high risk module according to the users may not be treated the same by the producer. This paper presents a method to determine the *producer risk index* for quantifying the probability and impact of failure on the producer at the product release time. The producer can use the results to determine whether it is more economical to improve the software quality or to release the product with the achieved quality level. The index can also be used to estimate the maintenance effort. The approach makes use of two recent developments in software complexity metrics, principal component analysis and reliability growth modelling. A new classification scheme for failure impact is introduced. Results from applying the producer risk index to a telecommunication software system are presented.

KEY WORDS: producer risk index; risk assessment; metric

1. INTRODUCTION

Software systems are being deployed in a wide range of critical applications that have severe failure implications. These applications include transport, energy, commercial, finance and military systems. Failures of these systems may cause loss of life, loss of critical information, and loss of property.

Software risk management has been receiving increasing attention recently (Littlewood and Strigni, 1992). Risk management consists of two processes: risk assessment and risk control. The risk assessment process includes risk identification, risk analysis and risk prioritization steps while the risk control process is composed of risk management planning, risk resolution and risk monitoring steps (Boehm, 1989).

The key risk factors at the time of product release are insufficient content and unsatisfactory quality of the software. Sometimes the software producer must deliver the product on time. All too often either the content or the quality of the software product will suffer. The former deficiency can be easily determined and eliminated if the software satisfies the functionalities required as stated in its specification. For improved quality, more testing can be done to try to detect more faults. As expected, reducing risk almost always involves additional expense.

In risk analysis, the risk of a software product is quantified. Some attempt of defining

CCC 1040–550X/96/050281–14

Received 1 September 1995
Revised 1 July 1996

a risk index has been made (Lee and Gunn, 1993; Leung, 1995). Previous attempts to define the risk index have taken the users' viewpoint. The loss due to a product failure is computed based on the impact on the users. We term this type of index the *user risk index.*

Users' expectation of the software is usually different from that of the producer. The users are more interested in the functionalities, quality and reliability of the software, delivery timescale and price. In terms of the software risk, the users are concerned with the mean time between failures of the product. Whenever a failure occurs, it means a 'down time' of the users' business. On the other hand, the software producer is concerned with the cost and profit generated from the project. This translates into concern over the number of latent defects in the product since these defects affect the maintenance support required for the product.

This paper presents a method to compute a *producer risk index* to quantify the risk to the producer at the product release time. The method first estimates the probability of failure and then estimates the failure impact on the producer. A new failure classification model is introduced. Failures are classified into several types based on the cost of fixing the failures. The model assumes that the major variation in the cost of fixing a failure is due to the number of designers involved in fixing the fault. Failures of different failure types are estimated using the source code complexity analysis, principal component analysis and regression analysis.

The producer risk index can be used for the following purposes:

- determining the impact on the maintenance effort,
- determining a cost effective amount of effort for improving the software quality,
- developing a business case to perform additional testing and delay the product release, if necessary.

The risk index assists development managers in making key product release decisions. Rather than faced with many metric data, development managers are provided with a single metric which represents an important attribute of software. The risk index partially eliminates the need of metric experts to interpret the many metric values such as code complexity, failure count and testing results in order to quantify the risk of a software system. Since there is a lack of metric experts in most companies, an automatic method for determining the risk of a system is beneficial to software management.

The paper is organized as follows. Section 2 gives the definition of producer risk index and Section 3 describes a new approach to computing the producer risk index. Section 3.1 presents a method for estimating the probability of failure and Section 3.2 gives a method to estimate the cost impact of a failure. Section 3.2.1 presents a model for resolving customer reported failures and analysing cost factors for fixing a failure. Section 3.2.2 describes a classification of faults according to the impact on the producer's effort for fixing the fault. Section 3.2.3 describes the computation of the failure impact. We demonstrate the feasibility of the approach using a telecommunication software system and selected results are presented in Section 4. Section 5 concludes with our plan for future research.

2. DEFINING THE PRODUCER RISK INDEX

Traditionally, the risk of a software module is defined as the likelihood of a failure occurrence and the consequence of that occurrence on the users. A *failure* is an incorrect output which is caused by a *fault* which is a programming mistake. For the sake of simplification of analysis, we assume that there is a one-to-one correspondence between fault and failure in this study. The *user risk index* (URI) is defined by the following relationship:

$$\text{URI} = \text{Prob(failure)} \times \text{Impact}_u\text{(failure)} \tag{1}$$

where Prob(failure) is the probability of a failure and Impact_u(failure) is the loss affecting the users if the failure occurs. Many factors contribute to the risks of software (Littlewood and Strigni, 1992). For example, change of specification, high complexity design, inability to test software thoroughly all increase the probability of failure.

Focusing on the loss affecting the software producer, we define the *producer risk index* (PRI) as follows:

$$\text{PRI} = \text{Prob(failure)} \times \text{Impact}_p\text{(failure)} \tag{2}$$

where Impact_p(failure) is the loss affecting the producer if the failure occurs.

Note that the risk index is a *product metric* which measures an external attribute (Fenton, 1991) of the software product with respect to how a failure of the product impacts its environment and the likelihood of such occurrence. The risk index is a predictive metric in that it tries to predict the future risk of using a software system.

Let U_H be the set of high risk modules according to URI and P_H be the set of high risk modules according to PRI. The users should aim to acquire systems with small U_H, as such systems represent lower risk. On the other hand, to reduce the maintenance effort, the producer should attempt to minimize P_H, as a large P_H set implies higher maintenance effort. In general

$$U_H \oplus P_H \neq \varnothing \tag{3}$$

where $\oplus$ represents the exclusive-or operator. In other words, there are modules which are considered as high risk from the users' view and are not considered as high risk from the producer's view, and vice versa. If the producer focuses on reducing only the risk of modules in P_H, then he will miss some modules which may have a serious impact on the users. Thus, to minimize the maintenance effort and satisfy the end users, a producer should improve the quality of modules in $U_H \cup P_H$.

3. COMPUTING THE PRODUCER RISK INDEX

The approach to computing the producer risk index is summarized in Figure 1. The required input includes the source code, failure and execution time data and failure type classification. The reliability analyser builds a reliability growth model with the failure and execution time data and then estimates the failure rate. The failure occurrence

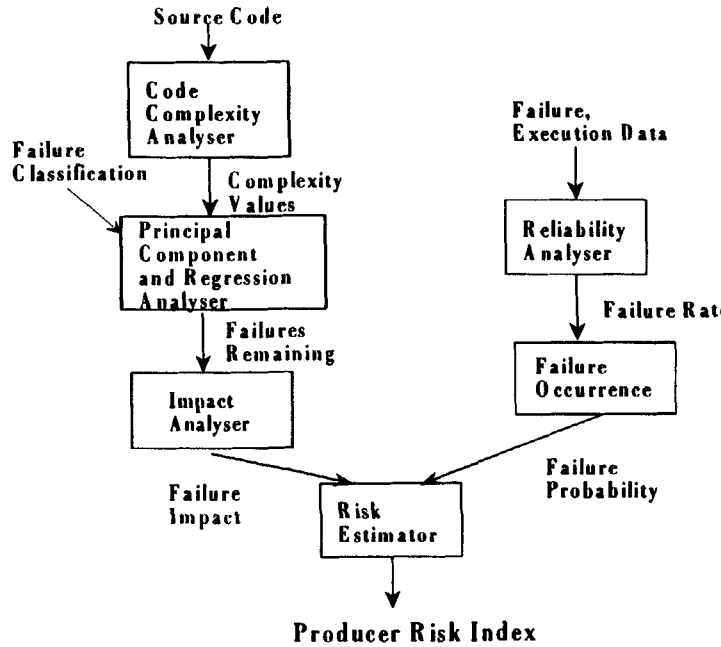

Figure 1. Producer risk index calculation

component simply computes the probability of failure using the method described in Section 3.1.

The input to the principal component and regression analyser are the set of code complexity metrics generated from a code complexity analyser. The principal component and regression analyser estimates the number of failures for each failure type. The impact analyser then estimates the failure impact. Section 3.2 presents the method to estimate the impact of failure of a software module.

3.1. Estimating probability of failure

We use the *predicted failure detection* approach to estimate the probability of encountering a failure during an arbitrary execution or use of the software system (Leung, 1995). Given the failure data and test cases execution time, a reliability growth execution time model such as the exponential and the logarithm Poisson execution time models may be used to predict the failure intensity for the next execution of the module (Musa, Ianino and Okumoto, 1987). To predict the probability of failure, we need to estimate the execution time of the next module execution which can be approximated with the average duration of an execution of module, d. We can determine d based on the number of executions, e, carried out during testing and the total execution time T accumulated up to the last execution.

$$d = T/e \tag{4}$$

Figure 2 plots the failure intensity against the execution time and illustrates the parameters used for computing the predicted failure intensity. Failure data up to time T is inputted into the reliability growth model. The failure intensity graph after time T represents the prediction from the reliability growth model. The predicted failure intensity n for the next execution of the software can be read off the graph at time $T+d$.

The probability of encountering a failure is the same as the predicted failure detection rate, r, at time $T+d$, which can be computed as

$$r = n \times T/e \tag{5}$$

3.2. Estimating failure impact on the producer

The loss or impact of a failure on the producer usually cannot be measured directly. As the intangible costs such as the loss of business and loss of productivity are hard to quantify, we will instead focus on the direct costs in fixing the fault in the following analysis. Studies have shown that the later in the software development life cycle that a software fault is detected, the higher is the fixing cost. It may cost 20 to 1 000 times as much to fix a *post-release fault* (a fault reported by the customer) compared with fixing a *prior-release fault* (a fault detected during development prior to the product release) (Boehm, 1981).

The costs of fixing different failures are not the same. For example, a failure which involves changes to many modules may require more effort than one which only requires changes to a single instruction. To estimate the impact of a failure, it is more accurate to identify its failure impact type and then use the cost associated with the failure impact type. In our approach, we first use the historical data, principal component analysis with code complexity analysis, and regression analysis to estimate the expected failures of each impact type, and then derive an average impact figure.

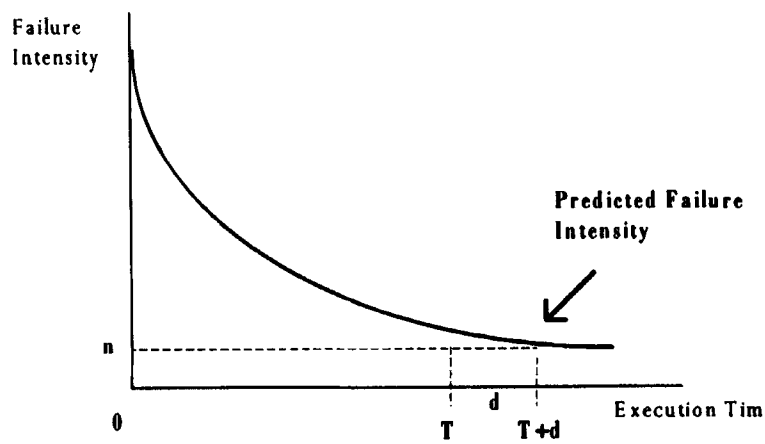

Figure 2. Failure intensity curve and predicted failure intensity

3.2.1. *A model for resolving customer reported failures*

This section presents a cost model for fixing a customer reported failure. In a typical scenario for handling a customer problem report, shown in Figure 3, the customer sends in the software problem report to the customer service department which first confirms the existence of the problem. It may request additional information from the customer, if the problem cannot be reproduced. There are three possible cases.

(1) The reported problem may be found not to be a 'real' problem. It may be caused by some misinterpretation of the requirement or information. The customer will be informed of this and no further action is taken.
(2) The reported problem is a real problem and it will be fixed in the next release. The customer will be informed of this and no further action is taken.
(3) The reported problem is a real problem and will be fixed in this release. The customer service department then directs the problem to the design group if no quick solution exists. The design group develops either a permanent or temporary fix for the problem. The fix is then routed to the test group which checks the solution prior to generating a patch.

For Cases 1 and 2, the cost to handle a failure involves only the customer service department. We will not treat these cases as the cost involved is small compared with the cost of fixing a failure.

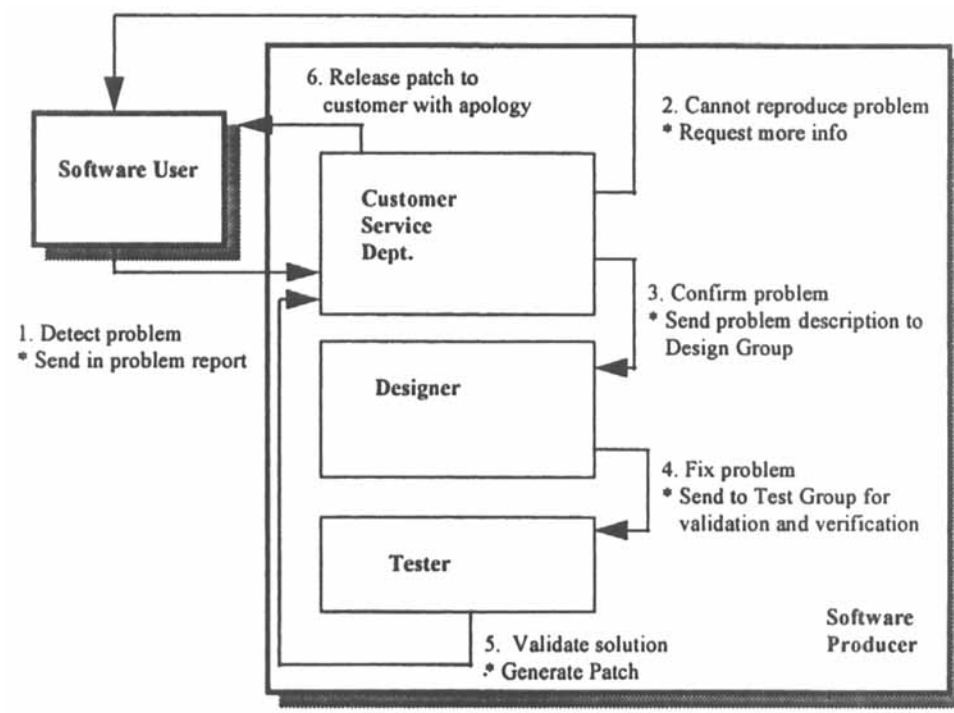

Figure 3. A scenario to handle a customer problem report

For Case 3 where a real problem is reported that requires corrective actions, we can identify the following post-release fix cost factors.

- Customer service department cost. This cost includes the time and resources for analysing the failure report, reproduce the failure, and request additional information from the customer, if needed.
- Designer fix cost. This includes the time and resources used by designers to fix the fault.
- Tester validation cost. This includes the time and resources used by testers to check that the fix actually resolves the customer-reported failure.
- Patch production cost. This includes the time and resources required to produce a new patch which contains fixes of the customer-reported failure.
- Patch release cost. This includes the time and resources required to release a new patch.

The impact on the producer for fixing a post-release failure is the sum of the five cost factors:

$$\text{Impact}_p = \text{Customer service department cost} + \text{Designer fix cost} + \text{Tester validation cost} + \text{Patch production cost} + \text{Patch release cost} \quad (6)$$

We next consider each of these cost factors. We will identify the cases when four of the five cost factors can be treated as constants and then develop a model relating the designer fix costs to failure types.

For a mature software development organization where standardized processes are in place, the customer service department cost, patch production cost and patch release cost are relatively independent of the faults because the effort and procedures involved for fixing one fault should be approximately the same as another.

Some organizations have a standard set of regression test cases that are run to validate any modification to their software systems. Quite often, these regression test cases have been automated and can be applied without any human supervision. If the regression testing package is standardized, the regression testing effort will be the same for all faults. Thus, the tester validation cost can be independent of the faults. We will call the customer service department, patch production, patch release and tester validation cost *fault-insensitive cost factors.*

The designer fix cost consists of two components: *analysis cost* and *change cost.* The analysis cost includes the time and resources required to study and isolate the fault to the code containing the fault and then develp a fix. The change cost includes the time and resources required to identify and modify the specific code to eliminate the fault.

Our analysis indicates that the designer fix cost is highly dependent on individual failures. To determine the basic underlying factors which contribute significantly to the designer fix cost, we collected data such as the number of designers involved in fixing a fault, the size and type of source code involved, the effort spent on fixing a fault, and the elapse time for resolving a fault. Based on the empirical data collected from four projects, the most important factor in determining the designer fix cost seems to be the number of designers involved in analysing and developing a resolution for the fault. Although we do not have sufficient data to establish a precise relationship, our preliminary

data suggest that as the number of designers involved in the fix increases, the designer fix cost also increases. This may be caused by the increase in the communication overhead, the higher chance of misunderstanding, and the difficulty in co-ordinating the work of many designers. In our cost model, we attempt to classify failures by the degree of involvement of designers in developing a fix for the failure. We accomplish this by relating failures to the source code required to be studied and changed for fixing the failures.

3.2.2. *Failure classification*

In this section, we develop a classification of failures based on the designer cost of fixing the failures. Let the *fault containing code* be the source code which contains the fault, and the *change code* be the source code that needs to be modified to eliminate the fault. Note that the fault containing code may involve several components of the software and they can be developed by different designers. Similarly the change code may involve several designers. Although the change code includes the fault containing code, it may include additional codes. For example, a fix may require changes to all modules which use a particular variable, even though the fault is present in a single module. Also, the fault containing code is not necessarily a subset of the change code.

In our case study, we group the fault containing code into two types:

A1: fault containing code developed by one designer
A2: fault containing code developed by more than one designer

Similarly, we classify the change code into two types:

C1: change code needs to be changed by one designer
C2: change code needs to be changed by more than one designer

We then classify the source code which needs to be studied and changed for fixing a customer reported failure into four types, as shown in Table 1.

A failure can then be classified according to the source code involved in developing a fix. Table 1 also shows the relationship between the source code type and failure type.

We can create more than four failure types to allow for a finer classification of designer fix costs. This is not done because a finer classification requires more data collection and analysis which may not be practical. Also, the extra failure types may not significantly improve the accuracy of impact estimate of our model. Note that the method used for calculating the producer risk index does not depend on the number of failure types.

Table 1. Source code type and failure type

Source code type	Code combination	Failure type
1	A1–C1	I
2	A1–C2	II
3	A2–C1	III
4	A2–C2	IV

In summary, the impact on the producer for a post-release failure can be estimated as

$$\text{Impact}_p = \text{Fault-insensitive cost} + \text{Designer fix cost} \tag{7}$$

where the first term is relatively independent of the failure type, and the second term depends on the failure type.

The cost model is based on the following postulates.

(1) The producer has standardized processes for the customer servicing department in resolving reported failures, for patch production and for patch release.
(2) A standard regression testing package is applied for testing any software modification.
(3) The designer fix cost depends on the failure type.

3.2.3. *Failure impact*

Let I_i denote the cost impact and D_i denote the number of failures of type i remaining in the software product. If all D_i, $i = 1$ to k, where k denotes the number of failure types, are known, then the average impact for fixing a failure (AI) is

$$AI = \sum_{i=1}^{k} I_i \times D_i / \sum_{i=1}^{k} D_i \tag{8}$$

We can increase the accuracy of the impact estimate by using more failure types. For the highest accuracy, we can use as many failure types as failures. However, using many failure types is not practical as it requires detail tracking of the cost of fixing each failure and thus a large effort for data collection. In our case study, we model four types of failures to reduce the data collection and analysis effort.

To determine the average impact for fixing a failure, we need a way to predict failures. Software complexity metrics may be used as predictors of failures (Khoshgoftaar and Munson, 1990; Munson and Khoshgoftaar, 1992). Recently, the principal component analysis applied to code complexity metrics has been shown to be able to identify error-prone modules (Khoshgoftaar and Munson, 1990; Munson and Khoshgoftaar, 1992). We modify the methodology introduced in (Khoshgoftaar, Munson and Lanning, 1993) to predict future failures in a module, rather than software changes.

The methodology first applies the principal component analysis technique (Dillion and Goldstein, 1984) to the software complexity data to identify the underlying orthogonal metric domains implied by the correlation of the complexity metrics. The orthogonal domain metrics that represent each distinct source of variance in the complexity data are derived next. The cluster analysis is then applied to the modules to partition these modules into clusters having similar complexity along the domain metric dimensions. We then develop within-cluster regression models using the derived domain metrics as independent variables and the number of failures as the dependent variable. In this study, we have selected the stepwise regression and backward elimination technique. More details on regression model selection can be found in Myers (1990). We carry out the above analysis to generate the failure prediction for each failure impact type.

4. APPLYING THE PRODUCER RISK INDEX

We have applied the producer risk index to a telecommunication software system. The software system is developed with a Pascal-like programming language. Each module is developed by one designer, using a company-wide software development process and environment. The source code after system test is used in the code complexity analysis. The failure data represent all failures detected during unit, integration and system test.

In this experimentation, we classified failures into four types, as outlined in Section 3.2.2. The impact of a failure type I is viewed as a baseline of unit cost. From the preliminary data, the costs of other failure types are as follow:

Type II : 1·1 × Type I
Type III: 1·1 × Type I
Type IV: 1·3 × Type I

For the computation of code complexity, we have chosen Datrix (DATRIX, 1990) because it is available internally for our use and it provides a comprehensive set of control-flow metrics. Datrix is a code complexity analyser originally developed in Bell Canada and is currently available commercially. It can analyse source code written in languages such as C, Pascal, and C++. Table 2 shows the metrics used in the regression analysis and failure prediction model.

Table 2. Metrics used

Metric
Breach of structure
Weighted breach of structure
Number of arcs
Number of crossing arcs
Mean identifier length
Number of instructions
Total number of lines
Mean decision summit complex
Number of decision summits
Number of exit summits
Mean nesting level
Weighted nesting level
Number of loops
Number of independent paths
Number of recursive summits
Mean decision summit span
Code arc comments ratio
Weighted loop ratio
Commented control structure ratio
Comments volume ratio
Number of summits
Comments volume of declaration
Comments volume of structures
Cyclomatic number
Structural volume

A regression model is developed using data from a *validation set* consisting of 100 modules. These modules total 151 000 lines of source code. The model is then used to provide failure prediction for an *application set* of 20 modules which are different from the validation set but belong to the same project. The failure prediction is generated separately for each failure type. The producer risk index is then calculated for modules of the application set.

This section focuses on the results of the computation for the producer risk index, rather than the results of the principal component analysis, clustering analysis and the regression modelling, since the steps involved can be found in other papers (Khoshgoftaar, Munson and Lanning, 1993; Leung, 1995). From the clustering analysis, modules of the validation set are coalesced to two clusters, A and B. Cluster A has 55 modules while cluster B has 45 modules. In the failure prediction analysis for the application set, each module is coalesced to the cluster having the closest centre. 12 modules coalesced to cluster A and the remaining 8 modules coalesced to cluster B. We then develop within-cluster regression models that predict software failures for modules of specific types based upon the domain metric values.

Table 3 shows the various data for the computation of the producer risk index. We list the module size in terms of thousand lines of non-commented source code (KLOC) in column 2. Column 3 provides the probability of failure of a module. Columns 4, 5, 6, 7 and 8 give the predicted failures from the regression model.

Module 5 has the highest predicted failure. This confirms the results from the reliability analysis which indicates that it does not exhibit any reliability growth. Since its

Table 3. Producer risk index of the application set

1 Module	2 Size (KLOC)	3 Prob (failure)	4 Failure I	5 Failure II	6 Failure III	7 Failure IV	8 Total predicted failure	9 AI	10 PRI
1	1·7	0·025	1	1	0	0	2	1·050	0·0263
2	1·4	0·040	2	0	0	0	2	1·000	0·0400
3	0·4	0·018	1	1	0	0	2	1·050	0·0189
4	0·7	0·005	0	1	0	0	1	1·100	0·0055
5	1·2	?	6	3	3	0	12	1·050	?
6	2·6	0·056	1	2	1	0	4	1·075	0·0602
7	0·6	0·051	2	1	1	0	4	1·050	0·0540
8	0·5	0·024	2	1	0	0	3	1·033	0·0248
9	1·1	0·013	0	1	0	0	1	1·100	0·0146
10	1·4	0·012	0	1	0	0	1	1·100	0·0132
11	1·2	0·007	0	1	0	0	1	1·100	0·0074
12	0·8	0·103	3	2	1	0	6	1·050	0·1083
13	3·1	0·240	2	2	0	1	5	1·100	0·2640
14	0·6	0·125	4	1	1	0	6	1·033	0·1292
15	3	0·007	1	0	0	0	1	1·000	0·0070
16	0·7	0·047	1	1	1	0	3	1·067	0·0498
17	0·9	0·007	1	0	0	0	1	1·000	0·0067
18	0·2	0·011	1	0	0	0	1	1·000	0·0114
19	3·1	0·006	0	1	0	0	1	1·100	0·0066
20	0·3	0·007	1	0	0	0	1	1·000	0·0070

Prob(failure) cannot be computed due to the lack of reliability growth (indicated by a '?'), the PRI of module 5 cannot be computed either. We treat any module with an unknown PRI as a high risk module.

The average impact for fixing a future failure is given in column 9. Column 10 lists the producer risk index, computed by multiplying columns 3 and 9.

We group modules of the application set into three risk classes:

(1) high risk modules when $0{\cdot}1 < \text{PRI}$
(2) medium risk modules when $0{\cdot}01 \leqslant \text{PRI} \leqslant 0{\cdot}1$
(3) low risk modules when $\text{PRI} < 0{\cdot}01$

In classifying modules into risk classes, we have chosen cut-off points that are the same as those used in the study of the user risk index (Leung, 1995). The cut-off points were chosen from the distribution of URI of the validation set. Approximately 20 per cent of the modules in the validation set have risk indexes higher than 0·1 and 20 per cent of the modules have risk indexes below 0·01. More data are needed to establish the usefulness of these cut-off points. A long term goal of this research is to determine the two cut-off points separating the high and medium risk, and medium and low risk modules. We expect data collected over the next year to provide answers.

According to the above classification scheme, there are four high risk modules, ten medium risk modules and six low risk modules in the application set. All high risk modules have more than five predicted failures, and an average failure impact of around 1·06. Both properties suggest that they are likely to have failures and these failures will require extra effort to fix.

Compared with our previous results of the URI using the same software data (Leung, 1995), there are more high risk modules according to the URI. In the previous study, $|U_H| = 7$, including modules, 6, 7 and 16, which are not in P_H. This confirms that some high risk modules according to the users may not be classified the same according to the producer. More case studies are needed to check whether there are cases where the PRI will identify additional high risk modules that are not identified by the URI. A producer may choose to fix only the high risk modules in P_H, and miss some modules whose failure will have a high impact on the users. A producer who is 'customer-focused' should also consider modules in U_H when trying to improve software quality.

5. CONCLUSIONS

We have previously introduced a user risk index to quantify the risk to the end users when using a software module and a software product. The URI can be used by the users to estimate the risk of accepting and deploying a software system. This paper extends the approach to compute the producer risk index. The PRI can be used by the software producer for estimating the maintenance effort and thus can be used to motivate the producer to improve the product quality prior to product release. Our approach is both implementable and pragmatic. The risk index analysis is applied to a telecommunication software and has shown to be beneficial in identifying high, medium and low risk modules from the producer's perspective.

The estimates of the failure impact are based on the best information we have at the time and are viewed as 'reasonable' by the development team. These estimates need to

be further validated. Although the data collected seem to suggest that the fix cost ratios of failure Type I: II: III: IV are 1: 1·1: 1·1: 1·3, additional data are needed to confirm that this is the case for the specific development environment under study. We are currently collecting additional data to confirm the correlation between the failure type and impact on the software producer. It is likely that the fix cost ratios will depend on the development environment and vary from organization to organization.

While short of giving a definite risk measurement, our approach provides a way to compare the relative risk of different modules. If the cost of each failure type can be determined, our approach can quantify the software risk.

We intend to carry out a validation of the risk index by checking its predictive power in identifying high risk modules. We will track the high risk modules of the application set and determine whether they indeed have failures which cause a higher impact than other modules. We expect data collected over the next two years to provide answers.

Acknowledgement

This research is supported by the Hong Kong Polytechnic University Research Grant 353-030.

References

Boehm, B. W. (1981) *Software Engineering Economics*, McGraw Hill, New York, NY.

Boehm, B. W. (1989) *Software Risk Management*, IEEE Computer Society Press, Los Alamitos, CA, U.S.A.

DATRIX (1990) *Datrix User's Manual*, Schemacode International Inc, Canada.

Dillion, W. R. and Goldstein, M. (1984) *Multivariate Analysis—Methods and Applications*. Series in Prob. and Math. Statistics, Wiley, New York.

Fenton, N. (1991) 'Software measurement: why a formal approach?', in *Formal Aspects of Measurement*, Denvir, T., Herman, R. and Whitty, R. W. (Eds), Springer-Verlag, London, pp. 5–27.

Khoshgoftaar, T. M. and Munson, J. C. (1990) 'Predicting software development errors using complexity metrics', *IEEE Journal of Selected Areas in Communications*, **8**(2), 253–261.

Khoshgoftaar, T. M., Munson, J. C. and Lanning, D. L. (1993) 'A comparative study of prediction models for program changes during system testing and maintenance', in *Proceedings Int. Conf. Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, U.S.A., pp. 72–79.

Lee, A. T. and Gunn, T. R. (1993) 'A quantitative risk assessment method for space flight software systems', in *Proceedings Fourth International Symposium of Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, U.S.A., pp. 246–252.

Leung, H. K. N. (1995) 'A practical risk index', in *Proceedings International Software Quality Management*, Computational Mechanics Publications, Southampton, U.K., pp. 215–226.

Littlewood, B. and Strigni, L. (1992) 'The risks of software', *Scientific American*, November, pp. 62–75.

Munson, J. C. and Khoshgoftaar, T. M. (1992) 'The detection of fault-prone programs', *IEEE Trans. Software Eng.*, **SE-18**(5), 432–433.

Musa, J. D., Ianino, A. and Okumoto, K. (1987) *Software Reliability: Measurement, Predictions, Application*, McGraw Hill, Singapore.

Myers, R. H. (1990) *Classical and Modern Regression with Applications*, Duxbury Press, Boston, MA.

Author's biography:

Hareton Leung has conducted research in software testing, software maintenance, quality and process improvement, and software metrics. He gained practical experience with software development while working for Bell-Northern Research and Northern Telecom. Dr. Leung is with the Department

of Computing of The Hong Kong Polytechnic University and serves as the Associate Director of the *Software Technology Facilities Center*, which focuses on software technology transfer, and process and quality improvement research. He is a founding member of *Hong Kong Software Process Improvement Network (HKSPIN)*. Dr. Leung serves on the Editorial Board of *IEEE Computer Society Press—Computer Science and Engineering Practice Press* and Program Committee of several international conferences. Dr. Leung earned a BSc degree in physics and astronomy from the University of British Columbia, a Master in CS from Simon Fraser University, and a PhD in CS from the University of Alberta.